Dependability Improvement Depends on Dependable Measurement

Les Hatton, *Emeritus Professor, Kingston University, London,*

Abstract—In this review of dependable computing I will try to embrace not only the tireless efforts of computer scientists everywhere working to tame this phenomenal technology of Computing, but also the perspective from the much greater number of people who have to depend on it for their pleasure, their living and in some cases, their lives. If computing is ever to be considered dependable by all its stake-holders however, we must address the most fundamental aspect of process improvement, that of measurement. As currently practised, it is wholly and demonstrably inadequate.

Index Terms—Computer Society, IEEE, Dependable Computing, journal, LaTeX, paper, template.

1 Introduction

Even a brief review of the Topic of Dependable Computing covers a vast range of avenues and as such I have created references to both this paper and a bibliography in Supplementary Materials. Bibliography references may be found in Supplementary Materials and are prefixed by SM.

It is clear that researchers have been worrying about dependability for a long time [SM1], [SM2], [SM3], [SM4], [SM5], [SM6]. Indeed the need for something as seemingly mundane as the desk checking of a program appears to have been recognised by Alan Turing in the late 1940s. Second, as the years have gone by, more and more aspects of computing have become embroiled in the notion of dependability as our computing needs have become more sophisticated and more ubiquitous. Today, we must embrace at least the notions of economics as unreliability is expensive [1]; of safety as computers dominate the world of transport amongst others [2], [3]; of security as computing is by definition not dependable if its actions are in control of a malignant party [4]; and indeed of mental health as the frequently poisonous world of social media is a computational phenomenon. Finally, we have yet another elephant in the room, that of AI and the expected tsunami of auto-generated software. It is therefore not unreasonable for the would be builder of dependable systems to interpret dependable as protecting from harm in a broad sense.

The end-user will have an even broader understanding of dependable, and will have in mind an experience close to the legal and indeed self-evident concept of *quiet ownership*. Of course they don't want to be harmed but they want the product they buy and the frequently huge amount of software embedded in it, to perform largely as advertised for a suitably long period give or take the occasional reboot. Anything less than this cannot be considered dependable from their point of view.

It might seem a little odd to the uninformed reader that computer scientists even worry about making computing dependable. Why on earth would we have computer systems which were not dependable? After all, we do not have the same mindset with conventional engineering. It would be unconscionable to design a bridge which was not dependable for example, even if we occasionally do. It is true that there was a time in the distant past when civil and mechanical engineering were not dependable but by and large we have learned from our mistakes over the centuries these activities have taken place [SM7], [SM8], [SM9], although even now as demonstrated by the lamentable Boeing 737 MAX saga, sometimes those hard-won lessons and reputations are forgotten when we attempt to transfer those lessons to IT.

Unfortunately the concept of non-dependable computing has been thrust upon us by its track record of unreliability. Indeed, program crashes, bugs, unexpected behaviour, slow responses and a rapid overturn of technologies with strange acronyms which do not seem to improve matters, have become part of the landscape of dealing with computing. To the end user, they are at least incomprehensible, inconvenient and often worse especially as modern embedded control systems intrude into every aspect of life. Having to reboot your washing machine by unplugging it, giving it a few seconds to recover from the excitement and plugging it back in again is bad enough, but having to pay an engineer to discover this merely rubs salt into the wound. Many devices even have a little reset button, fairly well hidden as if the manufacturer is reluctant to include this. To the computer scientist of course this should be frankly embarrassing given the efforts over the years at attempting to improve the situation. Coincidentally, the day before this article was submitted (28-Aug-2023), the National Air Traffic Control System in the UK collapsed without adequate backup systems so controllers had to enter the airline data by hand. the stated reason was that this was caused by a corrupt flight record[1]. If true, this is lamentable. The result was chaos for several days in European air space.

So what is so special about computing that means that

• *Les Hatton was with the Faculty of Science, Kingston University, London, KT1 2EE, U.K.*
E-mail: see http://www.leshatton.org/

Manuscript received XXXXX DD, YYYY; revised XXXXXX DD, YYYY.

1. https://www.bbc.co.uk/news/uk-66644369, accessed 30-Aug-2023

neither computer scientists nor end-users can take its dependability as a given?

1.1 Early reliability

In the early days of computing, even the existence of a dependable underlying operating system could not be taken for granted as they usually can today, although there is invariably some excitement when an operating system transitions to a new version. They had to be designed and written from scratch, usually in assembler language, (Unix was the first operating system written in a higher-level language (C) in the late 60s and early 70s). Operating systems of course had to be dependable because they are a *sine qua non*. If the OS isn't dependable, neither is anything else. Amongst the leaders in developing these were IBM, Sperry, CDC, DEC and ICL. The IBM experience even spawned one of the best books ever written on the trials and tribulations of developing dependable programs, Fred Brooks' eponymous "Mythical Man Month", [SM10]. Glancing through this 48 year old masterclass today, it is reasonable to ask if we have actually learned anything.

The operating systems of the 60s, 70s and 80s were dependable in the sense of not failing very often. There isn't much available in the way of data (there still isn't), but anecdotally and by personal experience, they behaved and remained in the background which is where an operating system should be. Then along came the PC. All of sudden, the operating system was not invisible. Instead we had the infamous and painfully frequent "Blue screen of death". Older users will remember that it was not at all uncommon to spend most of the working day watching the PC reboot, particularly after an "upgrade". Dependability was not at the forefront of those early PC program designers. They were trying to create a new kind of consumer computing, and they succeeded beyond anybody's wildest dreams. Instead of a mean time between failures of weeks, this had now dropped to hours but the end users put up with it because they were getting games and new visual means of interaction which were previously unknown - the perceived benefits were worth the considerably degraded dependability.

Today, the operating systems are generally dependable again in that we do not expect them to fail to behave as expected even though they occasionally do. What then of the applications which run on them?

Over the years, at least three distinct threads for attempting to improve dependability have emerged,

- Software and System Testing
- Software Process and Standards
- The hope for Formalism

They are not independent in practice but have independent philosophies.

1.2 Software and System Testing

Wired into software systems development since the early days, is the notion that testing offers us the possibility not only of improving a system's dependability but also of quantifying it. There are many such texts starting with early pioneers [SM11], [SM12], [SM13] with recent timely reviews of current practice [5] but the empirical impact is still not clear today in spite of Software testing being part of the Computer Science curriculum for at least 20 years [6].

Software testing is something of a Cinderella occupation and there is still misunderstanding as to what it entails. The common viewpoint is that Software Testing is about proving that a system functions correctly. In practice it is quite the opposite. The whole point is to prove that the system does *not* function correctly. In other words it is a falsification exercise not a verification exercise. Whichever we use however does not obscure the fact that testing software has a major problem. The number of potential inputs to any significant piece of software is so vast that only a tiny percentage of possibilities can be considered. The hope is that these intersect with the way the software will be used by its intended users but this is a non-trivial exercise to say the least.

1.3 Software Process and Standards

One of the most significant growth areas of the last few decades in attempting to promote dependability is in software process specification and its standardisation. The belief is that by codifying "good" practice and enforcing this by standardization in some way will promote the production of dependable systems. This appears self-evident and is an essential component of how, for example, modern bridge building addresses failures and how they should be avoided. Some 50 years ago following some very high profile bridge collapses, attention focused on a component known as a box-section girder. In short, they failed catastrophically in some conditions. This occurred sufficiently often that it led to independent enquiries in Australia [2], Germany and the UK, where the Merrison committee was convened[3]

The details of box-girder bridges are irrelevant here but we are interested in the response of the engineering community in mitigating their vulnerabilities. The Merrison committee set about understanding how such failures occurred and how they might be avoided, finishing with its highly influential final report in June 1973. Such reports have legal teeth and are influential in affecting a court's decision on liability if a comparable failure occurs. However notional compliance with a report is not of itself enough to guarantee that the lessons embodied within it have been followed and it is worth noting the following quotation by Sir Ian Merrison after this detailed analysis.

> *"No amount of writing of design codes and writing of contracts can in the end be guaranteed to prevent the results of stupidity, carelessness or incompetence. But one can do a great deal to discourage these vices and that must be done."*

Our efforts in software engineering have not been so distinguished in spite of enormous amounts of documentation defining what "good" practice was, (e.g. ISO 5055, ISO 12207, ISO 25010, ISO 90003, and many others). In spite of all of this, some 35 years after the Merrison report appeared,

2. www.parliament.vic.gov.au/papers/govpub/VPARL1971-72No2.pdf, accessed 19-Aug-2023.

3. https://www.istructe.org/resources/blog/learning-from-history-box-girder-bridges/, accessed 19-Aug-2023.

the Chinook helicopter crash of 2006 featured all three sins, stupidity, carelessness *and* incompetence, as described in an excoriating report by [SM14].

There is no excuse for the actions of those people and organisations named in this report and their contributions to this crash but it rather begs the question, what do we mean by "good" practice in a discipline such as software systems engineering which is almost devoid of measurement based evidence?

1.4 Safety and the hope for formalism

Given the many potential deficiencies of Software Testing, the idea of formally proving the behaviour of a computer program took hold in the last century with many adherents based on the pioneering work by Dijkstra, Hoare and others. However, the attractiveness and apparent self-evident nature of a methodology does not guarantee quantifiable benefits and as is usual in computing methodologies, there have been few published attempts to quantify any dependability benefits empirically. A notable exception was [7] who studied an air-traffic control system specified in a formal language Z and implemented in a programming language C. This revealed some intriguing patterns including the fact that more errors were found in the Z specification than were identified in the running code. The target system was relatively small by modern standards (around 200,000 lines of code - more of this measure later), so the scalability of the methods used is unknown.

It is probably reasonable to say that neither Testing nor Formalism have scaled sufficiently well to keep up with the inexorable growth of software systems, reported by [8] as close to 20% per year measured in source code terms. As well as this growth, software methodologies and programming languages continue to proliferate complicating the dependability argument further. It is impossible to improve a system if it is changing too quickly to apply measurement-based feedback because the feedback rapidly becomes irrelevant.

New technologies are often greeted with great enthusiasm by their proponents but such enthusiasm is usually about features and rarely about dependability and in truth they seem merely different rather than better.

What is missing of course is the most essential element of engineering improvement - measurement-based feedback, whereby quantifiable knowledge and understanding of past failures permits future occurrences of the same or similar failures to be mitigated or even avoided.

2 The pivotal role of measurement

It is fair to say that the state of software measurement has simply not moved on. Decades ago, researchers were lamenting that computer scientists simply do not experiment enough with [9] making the point that measurement in computer science was mostly characterised by its absence in comparison with conventional sciences. It is not just for want of trying. One of the most cited papers in computer science [10] describes an experimental measure of complexity and its apparent relationship to defects. It has been both lauded and criticized in the intervening years [11]. Unfortunately, nearly 50 years of experience with it appear to have revealed no systematic benefits and yet a web search for "Cyclomatic Complexity" reveals that it still features on many programming advice sites on the web, no doubt because we are little further on in understanding how software systems fail.

It is always useful to compare the state of measurement with other sciences as Tichy and co-workers did 25 years ago *loc. cit.* Let us consider a relatively recent candidate. 25 years ago we had little or nothing in the way of genetic measurement data. This position has been completely transformed in the intervening period and most importantly for this article, *the results have been openly distributed throughout the internet for anyone to access.* We have open access to the nucleotide distributions of sequenced genomes[4], the amino-acid sequences of proteins[5], including their post-translationally modified versions and also defect rates in the form of SNP (Single Nucleotide Polymorphism) databases[6] which describe occurrences of single nucleotide omissions, additions, transpositions and so on. All a researcher needs is decent internet access (the protein database alone consists of some 200 million proteins at the time of writing along with ancillary biochemical and other meta data) and a bit of programming experience. The formats are open, human readable and perl and python packages to read them proliferate. Moreover many of the most common analysis tools for gene-wrangling are already openly available. The entire discipline has an open access and collaboration policy.

In the same period, although it is true that software engineering has found ways to make some source code available through the transformational process of open source, (for example github), the situation with regard to defect measurement is by comparison, abysmal. Commercial defect histories are noteworthy by their absence and there is comparatively little other defect data which is directly useful. While we continue as we do, we are going nowhere by comparison.

What then is so difficult about engaging measurement-based feedback in the fight against software defect and its direct impact on system dependability?

2.1 Defects and frequency

2.1.1 The eponymous line of code

The first thing we must reluctantly accept is that the stalwart of source code measurement, the line of code, *has to be discarded*. There are two main resons:

- It is not well-defined as it bears the syntactical character of whatever programming language is in use. So we might have SLOC (Source lines of code), which is roughly what we see in a text editor and includes blank and comment lines; PPLOC (Pre-processed lines of code), relevant to programming languages with a pre-processing stage in the compiler such as C and C++, realising that this expands included files;

4. For example, the links at https://www.sanger.ac.uk/data/1000-genomes/, accessed 19-Aug-2023.

5. https://ftp.uniprot.org/pub/databases/uniprot/, accessed 19-Aug-2023

6. Extensive list at https://www.hgvs.org/central-mutation-snp-databases, accessed 19-Aug-2023.

XLOC (Executable lines of code), which are those lines which cause a compiler to generate executable code, probably the most reliable measure. Unfortunately, these are in decreasing order of accessibility so those who bother quote SLOC as being synonymous with "Lines of Code".
- It has no relevance to any useful concept in Information theory.

Some further comments are in order. The experience of my co-author Michiel van Genuchten and I in the *IEEE Software* Impact series which has been running since 2010 [12] is that most organisations actually have only the roughest idea of how much code they actually have and perhaps more importantly, how much actually contributes to the runtime behaviour of their systems. It is usually rounded to the nearest million or so, but rarely does anybody actually know to the extent required for measurement-based improvement, although we do know quite accurately how fast open source code is growing in projects measured in SLOC [8].

The second point above comes to the fore in the light of [13] who demonstrate that by using programming language tokens, a concept directly related to Hartley-Shannon information, emergent properties are clearly visible in large populations of source code which remain obscured if the cruder measure of lines of code is used. In fact it emerges that the length distribution of software components measured in programming language tokens follows a precisely predictable distribution characterised by a sharp linear rise followed by a very precise power-law, *independently of what the software does or the language used.* The Appendix contains more detail of this phenomenon and the reason for its representation independence.

The first step forward then is to dispense with lines of code of any kind and measure program size in language tokens. Only then will we have a reliable, representation-independent measure of size with a firm theoretical basis.

2.2 Defects

If anything, the measurement support for defects is even worse. In essence a defect is an error in a program which causes it to behave unexpectedly and no paper on dependability would be complete without some kind of road-map on how defects are or might be distributed in any software system. Defects are intimately related to dependable computing but the relationship is complex to say the least. It is perfectly possible to have a program full of defects which is completely dependable by virtue of the fact that the only exercised bits of the software do not contain the defects. On the other hand, it is also perfectly possible to have a program with very few defects which is not dependable by virtue of their location in a commonly executed program path. This is very likely when the programmer's view of how a program will be used is very different from the end-user's perspective for example.

Having said this, computer scientists have mostly agreed that reducing defects is a good idea even though it is widely appreciated that zero-defect is a pipe-dream at least with current knowledge. Let us be under no illusions: defects occur in all software systems, however careful we are. In a computer program, the programmer might make logic errors, or misuse the programming language, or use a feature of a programming language which is not well defined, or may even be working from incorrect specifications. Unfortunately, injecting defects is far simpler than detecting and removing them. As a result, the study of defects, their definition, their distribution and their impact, has been engaging software researchers for decades, [SM15], [SM16], [SM13], [SM17], [SM18], [SM19], [SM20], [SM21], [SM22], [SM23], [SM24], [SM25]. In spite of this immense intellectual effort, it is a moot point whether we can agree on much more than that defects are a factor in software dependability but in what sense and to what extent is still unknown.

Numerous attempts were made in the last century to relate defects to something obvious and easy to measure and lines of code was an obvious and easily accessible measure. This resulted in a wide variety of models none of which appear to have any lasting predictive value, [11]. Part of the problem here is that the measurement of defect is often maddeningly vague. Such datasets as there are, are noisy in the extreme or it is not clear whether researchers are consistent in what they view as a defect. Even in a mature ubiquitous system with a consistent policy of defect reporting, no useful relationships with the source code emerged other than that defects cluster in components [SM26], by which we mean sub-programs, functions or subroutines. This is not new - it has been previously reported by [SM15], [SM19]. Matters are even worse in a commercial environment, where admitting the possibility of a defect in a product is likely to upset the management and legal team, so comprehensive commercial openly accessible case studies are almost unknown.

2.3 A conflation of units

At the root of the problem of course is that there is no clear relationship between the presence of defects and the dependability of the system in which they are found as noted more than 20 years ago by [11]. Perhaps we should not be surprised. The presence of defects and their frequency is intimately related to static properties of the source code as evidenced by our obsession with lines of code. The attraction of this is that given the source code we might be able to predict the dependability of the code in the hands of the user. Unfortunately the user of course is blissfully ignorant of what the source code looks like and has a completely different perspective. They are only interested in measures such as how long between failures or the probability of a system failing when it is asked to do something. These are temporal or probabilistic dependability measures and not without their own problems. For example, a computer program can be astonishingly dependable with one set of inputs and the opposite with another set. Given that the space of all possible inputs is often enormous as noted earlier and impossible to test in any reasonable time-scale, it is easy to see we have a problem here also. Nevertheless, some progress has been made with prediction models [SM27], [SM28], [SM25], although scaleability and extrapolation remains a significant problem in the rapid and ever-changing landscape of software engineering techniques.

2.4 Accepting emergent rules

It is not all bad news of course. Everything is dependable provided its limits can be determined by appropriate

measurement and applied in future designs to avoid straying outside those limits. In civil engineering, dependable transferable rules of thumb underpinned by some kind of measurement are vital. These rules simply emerge from the behaviour of systems such as the propensity for box-section girders to fail within a civil engineering structure as discussed earlier.

We can ask if similar transferable rules have emerged from any of the software engineering experiments we have run. There certainly have been defect studies which contain such transferable lessons although qualitative rather than quantitative, for example Adams long-term study of the IBM operating system [SM29] which demonstrated that a significant percentage of defects took an exceedingly long time to fail for the first time. Theoretical work illustrating the difficulties that this presents to the analysis of dependability was carried out by Littlewood and co-workers [14], [15] and we must remember that one experiment does not make a rule.

Mimicking some aspects of conventional engineering such as redundancy using N-version methods has been successful and there is an extensive literature on this starting with [SM1]. There are still concerns about the independence of the individual channels as raised originally by [16] and revisited by [17] but the technology certainly appears to improve dependability and is used in for example, railway signalling [18]. The problem inhibiting its scaleability is cost with each channel being developed independently.

Given all these uncertainties, it seems unlikely that we will ever get a handle on a mechanistic view of how defects are introduced, much less on how they then go on to cause failure. In fact such a viewpoint may even be irrelevant as there is now evidence that at least some aspects of software systems such as their component length distribution as a function of programming language tokens are an emergent property of all discrete systems. (see Appendix).

2.5 Is AI going to help?

New technologies in IT still have the ability to titillate the IT industry in spite of the extraordinary number of technologies we have inflicted on the world in the last 25 years or so. AI is simply the latest in an everlasting stream of creativity. As always, prospective users must tread the tightrope between possible beneficial advantage and certainly in the case of AI, likely considerable dangers. This wouldn't be the first IT technology which promised much but turned out to have a significant downside. Social media promised a new form of informal interaction and easy sharing of opinions. Nobody really foresaw the downsides: internet bullying, the fuelling of conspiracy theories, anonymous trolling, the proliferation of fake news and the like. Its not the technology's fault or even its inventors. It is simply because like most IT developments, there is boundless scope for users to find unexpected uses.

The use of AI seems limited only by the imagination and ambition of its supporters but of particular relevance here is that of automatic code generation. This appears to be a real problem in the making[7]

7. For example https://www.techspot.com/news/91984-almost-30-percent-new-github-code-written-ai.html and a number of other sources, accessed 19-Aug-2023.

It is easy to be a Luddite but perhaps the only sensible comment worth making at this point is: *"Would you trust AI generated code trained on what we have managed to produce so far?"* This needs to be taken much more seriously.

2.6 Dependable computing and the future of science

This may sound somewhat apocalyptic but the scientific method itself is at something of a crossroads as more and more of its traditional landscape of empiricism and falsifiability is invaded by results derived from computer programs of essentially unquantifiable dependability, [19]. It is only relatively recently that some journals have started requiring the complete computational means of reproducing any and all results quoted in a particular paper. Most journals do not however and it is probably not unfair to state that far too much scientific research depending on extensive computation is wrong to an unquantifiable degree. Unpalatable though it may be, this will no doubt improve but it is symptomatic of reliance on an immature technology and dependable software is indeed an immature technology. For a good discussion of this see [20].

3 Conclusion

It would seem that measurements of software defects, failure rates and even something as basic as size are so primitive or non-existent as to completely undermine attempts to refine a technology by the time-honoured engineering methodology of measurement-based feedback. We can certainly do something about size as described in the Appendix but this is merely one facet and there is little hope of employing failure feedback whilst we operate with one hand tied behind our backs.

There are many reasons for this, and some would argue laudable reasons. For example, technological overturn occurs at an extraordinarily rapid rate in IT in general and in software development in particular with entire methodologies coming and going along with a seemingly inexhaustible supply of new languages. This certainly satisfies our creative instincts but means that no technology keeps still long enough for any analysis of failure measurements to exert its influence. This is in stark contrast not only with conventional engineering disciplines but even with computer hardware development where progress has been immense in the same period. One of the superficial advantages of not measuring anything of course is the "ignorance is bliss" principle. If we are to train generative AI systems to produce the next generation of code, don't expect them to remove the bugs. To an AI system they are merely data. We are the ones that define bugs and we have not been very good at it so far.

It may be fun but it isn't engineering.

Appendix A
Static Measures

A.1 The most likely distribution of defects

Earlier, it was stated that one of the key advantages of using programming language tokens rather than lines of code was its natural relationship with Hartley-Shannon Information [SM30], [SM31]. Essentially, tokens are the elements of programming language recognised at the lexical analysis stage

of a compiler. They are free of the arbitrariness of definition of lines of code and are uniquely-defined, indivisible and form part of an alphabet of symbols whose information content is well-defined. As such, they are an excellent candidate as a length measure but we need to justify that this is a worthwhile step given the increased difficulty in extracting tokens compared with lines of code.

Such justification is provided by [SM32], [13] who show that using tokens as a length measure allows the power of statistical mechanics to be used to make a reassuringly precise component length distribution measured in tokens, confirmed at various scales and for various programming languages, (for example, Figs 9 and 10 of [SM32]). At the larger scales, the predicted pdf given by Fig. 1 accurately anticipates the distribution of function lengths displayed as frequencies of occurrence in tokens of Fig. 2, taken from an aggregate of 80 million (or so) lines of code. The theory, which is based on the Conservation of Hartley-Shannon Information (CoHSI) [13], predicts an asymptotic power-law of high quality *whatever the implementation language, technology or application area.* for components larger than the mode of Fig. 1 and this is indeed what is found in the sense that large quantities of measurement data based on the availability of open source software, for example Fig. 2, are unable to falsify it.

As non-intuitive as it may sound, it is as true and indeed directly related to the extraordinary and widely-known power-law which was observed by [SM33] to predict word frequencies in texts irrespective of their language, author or subject. In a software context, power-laws in length distributions have the property of producing significant numbers of very large components in all systems because power-laws do not decay to zero very quickly [SM34]. The presence of large components in a system therefore is not necessarily a symptom of poor design decomposition but more likely a statistical inevitability. It is quite simply the most likely distribution.

Finally, to emphasize the commonality of this distribution amongst discrete systems, [13], [SM35] also show that this is precisely the same length distribution as the known proteome, (a collection of currently 200 million proteins with length measured in amino acids [SM36]). Software function length distributions and protein length distributions are in terms of Information Theory, indistinguishable.

It can be strongly argued therefore that tokens be used as a measure of software source code size rather than any attempt to define a line of code. Although lack of space prohibits expanding on this here, lines of code do not correlate particularly well with tokens, especially for smaller components. For the data of Fig. 2, an R analysis using lm() gives an adjusted R^2 = 0.88, with p $< 10^{-16}$ with tokens/SLOC $\simeq 6$. We can summarise by saying that tokens are unambiguous, easily verifiable and language agnostic and have the required relationship to well-established theory to give an excellent basis for measuring the size of a computer program.

Can we predict defect occurrence in the same way?

The short answer seems to be yes in that the methodology of divining defect distributions using Information Theory embedded within Statistical Mechanics is just as applicable but sadly the available defect data is so sparse

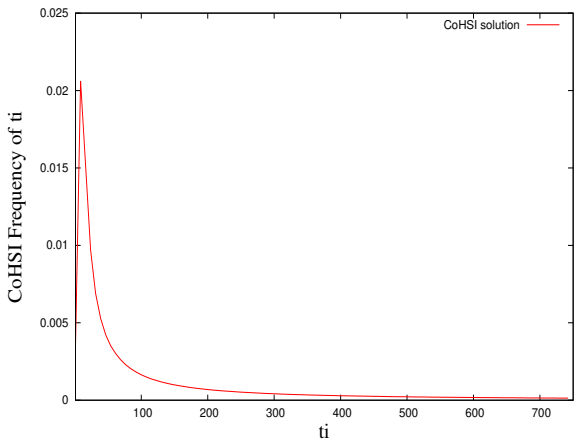

Fig. 1. A typical solution described by [13] shown as a pdf. Note the sharp unimodal peak followed for larger components by an extremely precise power-law.

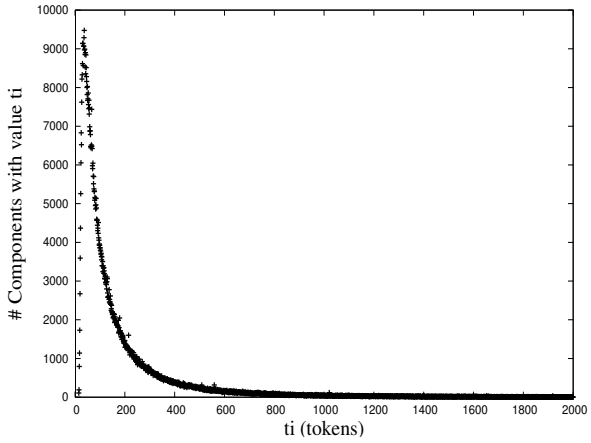

Fig. 2. 80 million lines of C shown as a frequency distribution of number of components versus size in programming tokens.

that no sensible attempt to falsify the resulting predictions can be made with current knowledge. There are one or two datasets of the required granularity with the required source code and defect data at the component level, for example, [SM22], [SM26], but this unfortunately is a drop in the ocean. Compared with the use of tokens above for which there are literally millions of data points, defect data is currently a non-starter.

This must change; a pipe-dream perhaps, but not as much a pipe-dream as expecting to improve a system by guesswork.

APPENDIX B
DYNAMIC MEASURES

Dynamic measures include failure rates and classification as well as probability of failure on demand. We know perfectly well how to measure these but judging by the paucity of widely available data, we either don't bother or we keep it secret. Neither is likely to help in any significant way.

ACKNOWLEDGMENTS

The author would like to thank the many researchers over the years who have attempted to understand this most intractable of problems.

REFERENCES

[1] Zhivich M, Cunningham R. The Real Cost of Software Errors. Security & Privacy, IEEE. 2009 05;7:87 90.
[2] Mun H, Han K, Lee DH. Ensuring Safety and Security in CAN-Based Automotive Embedded Systems: A Combination of Design Optimization and Secure Communication. IEEE Transactions on Vehicular Technology. 2020;69(7):7078-91.
[3] Koopman P. How Safe Is Safe Enough? Measuring and Predicting Autonomous Vehicle Safety. Independent; 2022.
[4] Fiondella L, Nikora A, Wandji T. Software Reliability and Security: Challenges and Crosscutting Themes. In: 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW); 2016. p. 55-6.
[5] Kassab M, DeFranco JF, Laplante PA. Software Testing: The State of the Practice. IEEE Software. 2017;34(5):46-52.
[6] Jones EL. Software testing in the computer science curriculum – a holistic approach. In: Proceeding ACSE '00 Proceedings of the Australasian conference on Computing education. New York, NY, USA: ACM; 2000. 10.1145/359369.359392.
[7] Pfleeger SL, Hatton L. Do formal methods really work ? IEEE Computer. 1997;30(2):p.33-43.
[8] Hatton L, Spinellis D, van Genuchten M. The long-term growth rate of evolving software: Empirical results and implications. Journal of Software: Evolution and Process. 2017;29(5).
[9] Tichy WF. Should computer scientists experiment more ? IEEE Computer. 1998 May;31(5):32-40.
[10] McCabe T. A software complexity measure. IEEE Transactions on Software Engineering. 1976;2(4):308-20.
[11] Fenton NE, Neil M. A critique of software defect prediction models. IEEE Transactions on Software Engineering. 1999;25(5):675-89.
[12] van Genuchten M, Hatton L. Software: What's in it and What's it in ? IEEE Software. 2010;27(1):14-6. Http://doi.ieeecomputersociety.org/10.1109/MS.2010.19.
[13] Hatton L, Warr G. Strong evidence of an information theoretical conservation principle linking all discrete systems. RSoc open sci. 2019 11;6(191101).
[14] Littlewood N, Strigini L. Validation of Ultra-High Dependability for Software-based Systems. Communications of the ACM. 1993;36(11):69-80.
[15] Littlewood B, Strigini L. Software reliability: basic concepts and assessment methods. Software Engineering, International Conference on. 2000;0:831. Http://doi.ieeecomputersociety.org/10.1109/ICSE.2000.10085.
[16] Knight JC, Leveson NG. An experimental evaluation of the assumption of independence in multi-version programming. IEEE Transactions on Software Engineering. 1986;12(1):96-109.
[17] Hatton L. Are N versions better than one good version ? IEEE Software. 1997;14(6):71-6.
[18] Eriş O, Yıldırım U, Durmuş MS, Söylemez MT, Kurtulan S. N-version Programming for Railway Interlocking Systems: Synchronization and Voting Strategy. IFAC Proceedings Volumes. 2012;45(24):177-80. 13th IFAC Symposium on Control in Transportation Systems.
[19] Ince DC, Hatton L, Graham-Cumming J. The case for open program code. Nature. 2012 02;482:485-8. Doi:10.1038/nature10836.
[20] Thimbleby H. Improving Science That Uses Code. The Computer Journal. 2023 08:bxad067.

PLACE PHOTO HERE

Les Hatton Les Hatton Ph.D. is a mathematician and emeritus professor of computing at Kingston University, London. He is currently working on problems in genetics.

SUPPLEMENTARY MATERIALS BIBLIOGRAPHY

[SM1] A. Avizienis and J.-C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, 1986.

[SM2] Les Hatton. *Safer C: Developing software in high-integrity and safety-critical systems*. McGraw-Hill, 1995. ISBN 0-07-707640-0.

[SM3] Hassan B. Diab and Albert Y. Zomaya. *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*. Wiley, 2005.

[SM4] John Knight. *Fundamentals of Dependable Computing for Software Engineers*. Routledge, 2012.

[SM5] Rogerio de Lemos. Special issue on dependable computing: theory and practice. *Computing*, 101:75–76, 2019.

[SM6] Long Wang. *Introduction: Software Dependability*, pages 3–5. Springer International Publishing, Cham, 2023.

[SM7] Henry Petroski. *To Engineer is Human: the role of failure in successful design*. Vintage, 1992.

[SM8] Charles Perrow. *Normal Accidents: living with high risk technologies*. Princeton University Press, 1999.

[SM9] Henry Petroski. *Success through Failure: the paradox of design*. Princeton University Press, 2008.

[SM10] F.P. Jnr. Brooks. *The Mythical Man Month*. Addison-Wesley, 1975. ISBN 0-201-00650-2.

[SM11] Glenford J. Myers. *The Art of Software Testing*. Wiley, 1979. isbn 978-0-471-04328-7.

[SM12] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, 10(2):14–32, March 1993. 10.1109/52.199724.

[SM13] B. Beizer. *Software Testing Techniques*. Van Nostrand, 1990. ISBN 0-442-20672-0.

[SM14] C. Haddon-Cave. An independent review into the broader issues surrounding the loss of the raf nimrod mr2 aircraft xv230 in afghanistan in 2006, Oct 2009. isbn 978-0-10-296265-9.

[SM15] V.R. Basili and B.T. Perricone. Software errors and complexity: an empirical investigation. *Comm. ACM*, 27(1):42–52, January 1984. http://www.lsmod.de/ bernhard/cvs/text/dipl/papers/p42-basili.pdf.

[SM16] B.T. Compton and C. Withrow. Prediction and control of Ada software defects. *Journal of Systems and Software*, 12:199–207, 1990.

[SM17] Jeff Tian and Joel Troster. A comparison of measurement and defect characteristics of new and legacy software systems. *Journal of Systems and Software*, 44(2):135 – 146, 1998. DOI: 10.1016/S0164-1212(98)10050-X.

[SM18] N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS, 2nd edition, 1997.

[SM19] B. Boehm and V.R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 34(1):135–137, 2001.

[SM20] R. Subramanyam and M.S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, April 2003.

[SM21] Hongfang Liu A. Gunes Koru, Dongsong Zhang. Modeling the effect of size on defect proneness for open-source software. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 10, Washington, DC, USA, 2007. IEEE Computer Society.

[SM22] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.

[SM23] A. Güneş Koru, Khaled El Emam, Dongsong Zhang, Hongfang Liu, and Divya Mathew. Theory of relative defect proneness. *Empirical Softw. Engg.*, 13(5):473–498, 2008.

[SM24] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[SM25] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia. A Developer Centered Bug Prediction Model. *IEEE Transactions on Software Engineering*, 99, 2017.

[SM26] Tim R. Hopkins and Les Hatton. Defect patterns and software metric correlations in a mature ubiquitous system. *arXiv*, 2019. arXiv:1912.04014.

[SM27] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005. http://doi.ieeecomputersociety.org/10.1109/TSE.2005.49.

[SM28] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540, New York, NY, USA, 2007. ACM. http://doi.acm.org/10.1145/1315245.1315311.

[SM29] E.N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[SM30] R.V.L. Hartley. Transmission of information. *Bell System Tech. Journal*, 7:535, 1928.

[SM31] C.E. Shannon. A mathematical theory of communication. *Bell System Tech. Journal*, 27:379–423, 07 1948.

[SM32] Les Hatton and Greg Warr. Information theory and the length distribution of all discrete systems. *arXiv*, 9 2017. http://arxiv.org/pdf/1709.01712 [q-bio.OT].

[SM33] George K. Zipf. *Psycho-Biology of Languages: an introduction to dynamic philology*. Houghton-Miflin, Boston MA, 1935.

[SM34] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2:1–2:26, October 2008.

[SM35] L. Hatton and G.W. Warr. *Exposing Nature's Bias: the Hidden Clockwork behind Society, Life and the Universe*. Bluespear Publishing, 2022. isbn 978-1-908-42204-0.

[SM36] SwissProt. The SwissProt release, 17-03, 2017. SwissProt http://www.uniprot.org/.